



# Dependability Analysis of Control Systems using SystemC and Statistical Model Checking

Van Chan Ngo, Axel Legay

**RESEARCH  
REPORT**

**N° 8762**

July 2015

Project-Team ESTASYS





## Dependability Analysis of Control Systems using SystemC and Statistical Model Checking

Van Chan Ngo, Axel Legay

Project-Team ESTASYS

Research Report n° 8762 — July 2015 — [22](#) pages

**Abstract:** Stochastic Petri nets are commonly used for modeling distributed systems in order to study their performance and dependability. This paper proposes a realization of stochastic Petri nets in SystemC for modeling large embedded control systems. Then statistical model checking is used to analyze the dependability of the constructed model. Our verification framework allows users to express a wide range of useful properties to be verified which is illustrated through a case study.

**Key-words:** SystemC, Statistical Model Checking, Formal Verification, Dependability Analysis, Petri Nets

**RESEARCH CENTRE  
RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu  
35042 Rennes Cedex

## Dependability Analysis of Control Systems using SystemC and Statistical Model Checking

**Résumé :** Petri nets stochastiques sont couramment utilisés pour la modélisation de systèmes distribués afin d'étudier leur performance et fiabilité. Cet article propose une réalisation de Petri nets stochastiques en SystemC pour la modélisation de grands systèmes de contrôle embarqués. Puis statistical model checking est utilisé pour analyser la fiabilité du modèle construit. Notre cadre de vérification permet aux utilisateurs d'exprimer une large gamme de propriétés utiles à vérifier qui est illustrée par une case-study.

**Mots-clés :** SystemC, Statistical Model Checking, Formal Verification, Dependability Analysis, Petri Nets

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Background</b>	<b>4</b>
2.1	The SystemC Language . . . . .	5
2.1.1	Language Features . . . . .	5
2.1.2	SystemC Simulation . . . . .	5
2.2	Statistical Model Checking . . . . .	7
<b>3</b>	<b>SMC for SystemC Models</b>	<b>7</b>
3.1	SystemC Model State . . . . .	7
3.2	Model and Execution Trace . . . . .	9
3.3	Expressing Properties . . . . .	9
<b>4</b>	<b>Implementation</b>	<b>10</b>
4.1	Monitor and Aspect-Advice Generator . . . . .	10
4.2	SystemC Plasma Lab Plugin . . . . .	11
4.3	Running Verification . . . . .	11
<b>5</b>	<b>Modeling Dependability in SystemC</b>	<b>11</b>
5.1	Stochastic High-Level Petri Nets . . . . .	11
5.2	Connection between HLPNs and Rule-Based Systems . . . . .	12
5.3	Realization of SHLPNs . . . . .	13
<b>6</b>	<b>Case Study and Results</b>	<b>15</b>
6.1	An Embedded Control System . . . . .	16
6.2	Analysis Results . . . . .	17
<b>7</b>	<b>Related Work and Conclusion</b>	<b>18</b>

## 1 Introduction

Computer-based control systems are increasingly used now in a wide range of industrial and military domains such as manufacturing, transport, energy and defense. In many cases, they are safety-critical systems (e.g., control systems for air-traffic, power plants, medical devices). Hence, it is necessary to quantify the probability or rate of all safety-related faults: How likely the system is available to meet a demand for service? What is the probability that the system repairs itself after a failure (e.g., the system conforms to existent and prominent standards such as the Safety Integrity Levels)? A general approach for performing such dependability analysis consists in constructing and analyzing a state-based model of the system [20, 8]. One of the main approaches, *Probabilistic Model Checking* (PMC), is an automatic technique for checking whether or not probabilistic models satisfy certain specifications, which is widely used to verify timed and probabilistic systems [11, 16]. One of the main challenges is the complexity of the algorithms in terms of execution time and memory space. Indeed, such algorithms suffer from the state space explosion problem, that is, the size of the state space tends to grow exponentially faster than the size of the system. As a result, the analysis of large systems is infeasible.

An alternative way to evaluate these systems is *Statistical Model Checking* (SMC), a simulation-based approach. Simulation-based approaches do not construct all the reachable states of the system-under-verification (SUV), thus they require far less execution time and memory space than numerical approaches. They have shown the advantages over other methods such as PMC on several case studies [15, 10].

In this work, we construct a SMC-based verification framework to analyze dependability of large industrial embedded control systems. Stochastic high-level Petri nets (SHLPNs) are traditionally used for modeling distributed control systems in order to study their performance and dependability. Therefore, we propose an approach to model the system dependability by realizing SHLPNs in SystemC.

We then analyze the constructed model in SystemC with Plasma Lab [2], a statistical model checker for stochastic processes, in which the properties to be verified are expressed in *Bounded Linear Temporal Logic* (BLTL). The implementation contains two main components: a *monitor generator* that instruments the SystemC model to generate the set of execution traces, and a *checker* that verifies the satisfaction of the properties based on the given set of execution traces. The monitor generation relies on the techniques proposed by Tabakov et al. [29] that provide a rich set of primitives for exposing different parts of the model state during a SystemC simulation.

The remainder of this paper is organized as follows: the next section introduces the SystemC modeling language and reviews the main features of SMC. We consider the execution traces of a SystemC model and the implementation of our verification framework in Section 3 and Section 4. An approach to model the dependability of computer-based control systems is proposed in Section 5. Section 6 illustrates the modeling approach and the verification procedure by a case study. The paper terminates with some related work, a conclusion and an outlook to some directions for future research.

## 2 Background

This section introduces the SystemC modeling language and reviews the main features of statistical model checking for stochastic processes.

## 2.1 The SystemC Language

SystemC<sup>1</sup> is a C++ library [9] providing primitives for modeling hardware and software systems at the level of transactions. Every SystemC model can be compiled with a standard C++ compiler to produce an executable program called executable specification. This specification is used to simulate the system behavior with the provided event-driven simulator.

### 2.1.1 Language Features

A SystemC model is hierarchical composition of modules (*sc\_module*). Modules are building blocks of SystemC design, they are like modules in Verilog [30], classes in C++. A module consists of an interface for communicating with other modules and a set of processes running concurrently to describe the functionality of the module. An interface contains ports (*sc\_port*), they are similar to the hardware pins. Modules are interconnected using either primitive channels (i.e., the signals, *sc\_signal*) or hierarchical channels via their ports. Channels are data containers that generate events in the simulation kernel whenever the contained data changes.

Processes are not hierarchical, so no process can call another process directly. A process is either a thread or a method. A thread process (*sc\_thread*) can suspend its execution by calling the library statement *wait* or any of its variants. When the execution is resumed, it will continue from that point. Threads run only once during the execution of the program and are not expected to terminate. On the other hand, a method process (*sc\_method*) cannot suspend its execution by calling *wait* and is expected to terminate. Thus, it only returns the control to the kernel when reaching the end of its body.

An event is an instance of the SystemC event class (*sc\_event*) whose occurrence triggers or resumes the execution of a process. All processes which are suspended by waiting for an event are resumed when this event occurs, we say that the event is notified. A module's process can be sensitive to a list of events. For example, a process may suspend itself and wait for a value change of a specific signal. Then, only this event occurrence can resume the execution of the process. In general, a process can wait for an event, a combination of events, or for an amount time to be resumed.

### 2.1.2 SystemC Simulation

In SystemC, integer values are used as discrete time model. The smallest quantum of time that can be represented is called *time resolution* meaning that any time value smaller than the time resolution will be rounded off. The available time resolutions are femtosecond, picosecond, nanosecond, microsecond, millisecond, and second. SystemC provides functions to set time resolution and declare a time object, for example, the following statements set the time resolution to 10 picosecond and create a time object *t1* representing 20 picoseconds.

```
1 | sc_set_time_resolution(10, SC_PS);
2 | sc_time t1(20, SC_PS); // SC_PS : picosecond
```

The SystemC simulator is an event-driven simulation [1, 22]. It establishes a hierarchical network of finite number of parallel communicating processes which under the supervision of the distinguished simulation kernel process. Only one process is dispatched by the scheduler to run at a time point, and the scheduler is non-preemptive, that is, the running process returns control to the kernel only when it finishes executing or explicitly suspends itself by calling *wait*. Like hardware modeling languages, the SystemC scheduler supports the notion of delta-cycles [19]. A delta-cycle lasts for an infinitesimal amount of time and is used to impose a partial order

---

<sup>1</sup>IEEE Standard 1666-2005

of simultaneous actions which interprets zero-delay semantics. Thus, the simulation time is not advanced when the scheduler processes a delta-cycle. During a delta-cycle, the scheduler executes actions in two phases: the *evaluate* and the *update* phases.

The simulation semantics of the SystemC scheduler is presented as follows: (1) *Initialize*. During the initialization, each process is executed once unless it is turned off by calling *dont\_initialize()*, or until a synchronization point (i.e., a *wait*) is reached. The order in which these processes are executed is unspecified; (2) *Evaluate*. The kernel starts a delta-cycle and run all processes that are ready to run one at a time. In this same phase a process can be made ready to run by an event notification; (3) *Update*. Execute any pending calls to *update()* resulting from calls to *request\_update()* in the evaluate phase. Note that a primitive channel uses *request\_update()* to have the kernel call its *update()* function after the execution of processes; (4) *Delta-cycle notification*. The kernel enters the delta notification phase where notified events trigger their dependent processes. Note that immediate notifications may make new processes runnable during step (2). If so the kernel loops back to step (2) and starts another evaluation phase and a new delta-cycle. It does not advance simulation time; (5) *Simulation-cycle notification*. If there are no more runnable processes, the kernel advances simulation time to the earliest pending timed notification. All processes sensitive to this event are triggered and the kernel loops back to step (2) and starts a new delta-cycle. This process is finished when all processes have terminated or the specified simulation time is passed. The simulation semantics can be represented by the pseudo code in Listing 1.

```

1 PC // All primitive channels
2 P // All processes
3 R = ∅ // Set of runnable processes
4 D = ∅ // Set of pending delta notifications
5 U = ∅ // Set of update requests
6 T = ∅ // Set of pending timed notifications
7 // Start elaboration: collect all update requests in U
8 for all chan ∈ U do
9   run chan.update()
10 end for
11 for all p ∈ P do
12   if p is initialized and p is not clocked thread then
13     R = R ∪ p // Make p runnable
14   end if
15 end for
16 for all p ∈ P do
17   if p is triggered by an event in D then
18     R = R ∪ p
19   end if
20 end for // End of initialization phase
21
22 repeat
23   while R ≠ ∅ do // New delta-cycle begins
24     for all r ∈ R do // Evaluation phase
25       R = R \ r
26       run r until it calls wait or returns
27     end for
28     for all chan ∈ U do // Update phase
29       run chan.update()
30     end for
31     for all p ∈ P do // Delta notification phase
32       if p is triggered by an event in D then
33         R = R ∪ p // Make p runnable
34       end if
35     end for // End of delta-cycle
36   end while
37
38   if T ≠ ∅ then
39     Advance the simulation clock to the earliest timed delay t
40     T = T \ t
41     for all p ∈ P do // Timed notification phase
42       if t triggers p then
43         R = R ∪ p // Make p runnable

```



```

44 |         end if
45 |     end for
46 | end if
47 | until end of simulation

```

Listing 1: Simulation semantics of SystemC

## 2.2 Statistical Model Checking

Let  $\mathcal{M}$  be a model of a stochastic process and  $\varphi$  be a property expressed as a BLTL formula. BLTL is a temporal logic with bounded temporal operators, ensuring that the satisfaction of a formula by a trace can be decided in a finite number of steps. The statistical probabilistic model checking problem consists in answering the following questions.

- *Qualitative.* Is the probability that  $\mathcal{M}$  satisfies  $\varphi$  greater or equal to a threshold  $\theta$  with a specific level of statistical confidence?
- *Quantitative.* What is the probability that  $\mathcal{M}$  satisfies  $\varphi$  with a specific level of statistical confidence?

They are denoted by  $\mathcal{M} \models Pr(\varphi)$  and  $\mathcal{M} \models Pr_{\geq \theta}(\varphi)$ , respectively.

The key idea of SMC [18] is to get, through simulation, a large amount of independent execution traces and count the number of traces that satisfy  $\varphi$ . The ratio of this number over the total number of execution traces provides the probability that the property holds. Then statistical results associate a level of confidence to this probability, depending on the number of execution traces. Many statistical methods including sequential hypothesis testing, Monte Carlo method, or 2-sided Chernoff bound are implemented in a set of existing tools [32, 2], that have shown their advantages over other methods such as PMC on several case studies.

Although SMC can only provide approximate results with a user-specified level of statistical confidence (as opposed to the exact results provided by standard probabilistic model checking method), it is compensated for by its better scalability and resource consumption. Since the models to be analyzed are often approximately known, an approximate result in the analysis of desired properties within specific bounds is quite acceptable. SMC has recently been applied in a wide range of research areas including software engineering (e.g., verification of critical embedded systems) [10], system biology, or medical area [15].

## 3 SMC for SystemC Models

This section illustrates the use of SMC for verifying a SystemC program exhibiting timed and probabilistic characteristics by showing that the operational semantics of the program is viewed as a stochastic process. The implementation of our SMC-based verification framework is considered as well.

### 3.1 SystemC Model State

Temporal logic formulas are interpreted over execution traces and traditionally a trace has been defined as a sequence of states in the execution of the model. Therefore before we can define an execution trace we need a precise definition of the state of a SystemC simulation. We are inspired by the definition of system state in [29], which consists of the state of the simulation kernel and the state of the SystemC model. We consider the external libraries as black boxes, meaning that their states are not exposed.

The state of the kernel contains the information about the current phase of the simulation and the SystemC events notified during the execution of the model. We denote the finite set of variables whose value domain represent the information about the current phase of the kernel by  $V_{ker}$ , and the finite set of variables whose value domain represent the event notification by  $V_{eve}$ .

The state of the SystemC model is the full state of the C++ code of all processes in the model, which includes the values of the variables, the location of the program counter, the call stack, and the status of the processes. We use  $V_{var}$ ,  $V_{loc}$ ,  $V_{sta}$ , and  $V_{proc}$  to denote the finite sets of variables whose value domains represent the values of the variables, the location of the program counter, the call stack and the status of the processes, respectively. Let  $V = \{v_0, \dots, v_{n-1}\} = \bigcup_k V_k$  with  $k \in \{ker, eve, var, loc, sta, proc\}$ , be a finite set of variables that takes values in a domain  $\mathbb{D}_X$ . A value in  $\mathbb{D}_X$  represents a state of a SystemC simulation.

We consider here some examples of variables in  $V$  that represent the state of the simulation kernel and the SystemC model. A system state can consists of the information about all locations before the execution of all statements that contain the devision operator “/” followed by zero or more spaces and the variable “a” in a SystemC model (e.g., the statement  $y = (x + 1) / a$ ). Let *Producer* and *Consumer* be two modules of a SystemC model. Assume that *Producer* has a function *send()* and *Consumer* has a function *receive()*. Two variables *send\_start* and *send\_done* can be defined as Boolean variables in  $V$  that hold the value *true* immediately before and after a call of the function *send()*, respectively to express the status of the call stack. Similarly, the variable *rcv* holds the value *true* immediately after a call of the function *receive()*. Assume again that *Producer* has an event named *write\_event*, to observe whenever this event is notified, we can use a variable *we\_notified* that holds the value *true* immediately when the event is notified. We will consider how these variables can be defined in our implementation of the verification framework in the next section.

We have discussed so far the state of a SystemC model execution. It remains to discuss how the semantics of the temporal operators is interpreted over the states in the execution of the model. That means how the states are sampled. The following definition gives the concept of *temporal resolution*, in which the states are evaluated only in instances in which the temporal resolution holds. It allows the user to set granularity of time.

**Temporal resolution** A temporal resolution  $\mathcal{T}_r$  is a finite set of Boolean expressions defined over  $V$  which specifies when the set of variables  $V$  is evaluated.

Temporal resolution can be used to define a more fine-grained model of time than a coarse-grained one provided by a cycle-based simulation. We call the expressions in  $\mathcal{T}_r$  *temporal events*. Whenever a temporal event is satisfied or the temporal event occurs,  $V$  is sampled. For example, assume that we want the set of variables to be sampled whenever at the end of simulation-cycle notification or immediately after the event *write\_event* is notified during a run of the model. Hence, we can define a temporal resolution as the following set  $\mathcal{T}_r = \{e_{dc}, we\_notified\}$ , where *e<sub>dc</sub>* and *we\_notified* are Boolean expressions that have the value *true* whenever the kernel phase is at the end of the simulation-cycle notification and the event *write\_event* notified, respectively.

We denote the set of occurrences of temporal events from  $\mathcal{T}_r$  along an execution of a SystemC model by  $\mathcal{T}_r^s$ , called a *temporal resolution set*. The value of a variable  $v \in V$  at an event occurrence  $e_c \in \mathcal{T}_r^s$  is defined by a mapping  $\xi_{val}^v : \mathcal{T}_r^s \rightarrow \mathbb{D}_v$ . Hence, the state of the SystemC model at  $e_c$  is defined by a tuple  $(\xi_{val}^{v_0}, \dots, \xi_{val}^{v_{n-1}})$ .

A mapping  $\xi_t : \mathcal{T}_r^s \rightarrow \mathcal{T}$  is called a *time event* that identifies the simulation time at each occurrence of an event from the temporal resolution. Hence, the set of time points which correspond to a temporal resolution set  $\mathcal{T}_r^s = \{e_{c_0}, \dots, e_{c_{N-1}}\}$ ,  $N \in \mathbb{N}$  is given as follows.

**Time tag** Given a temporal resolution set  $\mathcal{T}_r^s$ , the *time tag*  $\mathcal{T}$  corresponding to  $\mathcal{T}_r^s$  is a finite or infinite set of non-negative reals  $\{t_0, t_1, \dots, t_{N-1}\}$ , where  $t_{i+1} - t_i = \delta t_i \in \mathbb{R}_{\geq 0}$ ,  $t_i = \xi_t(e_{c_i})$ .

### 3.2 Model and Execution Trace

A SystemC model can be viewed as a hierarchical network of parallel communicating processes. Hence the execution of a SystemC model is an alternation of the control between the model's processes, the external libraries and the kernel process. The execution of the processes is supervised by the kernel process to concurrently update new values for the signals and variables w.r.t the cycle-based simulation. For example, given a set of runnable processes in a simulation-cycle, the kernel chooses one of them to execute first in a non-deterministic manner as described in the prior section.

Let  $V$  be the set of variables whose values represent the state of a SystemC model simulation. Our way to mix stochastic and non-deterministic characteristics consisting of assuming that, at any moment of time, the values variables in  $P \subseteq V$  are determined by a given probability distribution (i.e., from the probability distributions used in the model). The values of variables in  $V \setminus P$  are chosen in the non-deterministic manner of the simulation scheduler. At any given moment of time, it is allowed that the choice of the variables in  $V \setminus P$  might influence the distribution on the next values of variables in  $P$ .

Given a temporal resolution  $\mathcal{T}_r$  and its corresponding temporal resolution set along an execution of the model  $\mathcal{T}_r^s = \{e_{c_0}, \dots, e_{c_{N-1}}\}, N \in \mathbb{N}$ , the evaluation of  $V$  at the event occurrence  $e_{c_i}$  is defined by the tuple  $(\xi_{val}^{v_0}, \dots, \xi_{val}^{v_{n-1}})$ , or a state of the model, denoted by  $V(e_{c_i}) = (V(e_{c_i})(v_0), V(e_{c_i})(v_1), \dots, V(e_{c_i})(v_{n-1}))$ , where  $V(e_{c_i})(v_k) = \xi_{val}^{v_k}(e_{c_i})$  with  $k = 0, \dots, n-1$  is the value of the variable  $v_k$  at  $e_{c_i}$ . We denote the set of all possible evaluations by  $V_{\mathcal{T}_r^s} \subseteq \mathbb{D}_V$ , called the *state space* of the random variables in  $V$ . State changes are observed only at the moments of event occurrences. Hence, the operational semantics of a SystemC model is represented by a *stochastic process*  $\{(V(e_{c_i}), \xi_t(e_{c_i})), e_{c_i} \in \mathcal{T}_r^s\}_{i \in \mathbb{N}}$ , taking values in  $V_{\mathcal{T}_r^s} \times \mathbb{R}_{\geq 0}$  and indexed by the parameter  $e_{c_i}$ , which are event occurrences in the temporal resolution set  $\mathcal{T}_r^s$ . An execution trace is a realization of the stochastic process is given as follows.

**Execution trace** An execution trace of a SystemC model corresponding to a temporal resolution set  $\mathcal{T}_r^s = \{e_{c_0}, \dots, e_{c_{N-1}}\}, N \in \mathbb{N}$  is a sequence of states and event occurrence times, denoted by  $\omega = (s_0, t_0)(s_1, t_1) \dots (s_{N-1}, t_{N-1})$ , such that for each  $i \in 0, \dots, N-1$ ,  $s_i = V(e_{c_i})$  and  $t_i = \xi_t(e_{c_i})$ .

$N$  is the length of the execution, also denoted by  $|\omega|$ . We denote the prefix of  $\omega$  by  $\omega_k = (s_0, t_0), (s_1, t_1) \dots (s_k, t_k)$ , and the suffix by  $\omega^k = (s_k, t_k)(s_{k+1}, t_{k+1}) \dots (s_{N-1}, t_{N-1})$ .

Let  $V' \subseteq V$ , the *projection* of  $\omega$  on  $V'$ , denoted by  $\omega \downarrow_{V'}$ , is an execution trace such that  $|\omega \downarrow_{V'}| = |\omega|$  and  $\forall v \in V', \forall e_c \in \mathcal{T}_r^s, V'(e_c)(v) = V(e_c)(v)$ .

### 3.3 Expressing Properties

We recall the syntax and semantics of BLTL [28], an extension of Linear Temporal Logic (LTL) with time bounds on temporal operators. A BLTL formula  $\varphi$  is defined over a set of atomic propositions  $AP$  as in LTL. A BLTL formula is defined by the grammar  $\varphi ::= true | false | p \in AP | \varphi_1 \wedge \varphi_2 | \neg \varphi | \varphi_1 U_{\leq T} \varphi_2$ , where the time bound  $T$  is an amount of time or a number of states in the execution trace. The temporal modalities  $F$  (the “eventually”, sometimes in the future) and  $G$  (the “always”, from now on forever) can be derived from the “until”  $U$  as follows.

$$F_{\leq T} \varphi = true U_{\leq T} \varphi \text{ and } G_{\leq T} \varphi = \neg F_{\leq T} \neg \varphi$$

The semantics of BLTL is defined w.r.t execution traces of the model  $\mathcal{M}$ . Let  $\omega$  be an execution trace of  $\mathcal{M}$ , we denote by  $\omega \models \varphi$  that fact that  $\omega$  satisfies the BLTL formula  $\varphi$ .

- $\omega^k \models true$  and  $\omega^k \not\models false$

- $\omega^k \models p, p \in AP$  iff  $p \in L(s_k)$ , where  $L(s_k)$  is the set of atomic propositions which are *true* in state  $s_k$
- $\omega^k \models \varphi_1 \wedge \varphi_2$  iff  $\omega^k \models \varphi_1$  and  $\omega^k \models \varphi_2$
- $\omega^k \models \neg\varphi$  iff  $\omega^k \not\models \varphi$
- $\omega^k \models \varphi_1 U_{\leq T} \varphi_2$  iff there exists an integer  $i$  such that  $\omega^{k+i} \models \varphi_2$ ,  $\sum_{0 \leq j \leq i} (t_{k+j} - t_{k+j-1}) \leq T$ , and for each  $0 \leq j < i$ ,  $\omega^{k+j} \models \varphi_1$

In our framework, the set of atomic proposition  $AP$  consists of the predicates defined over the set of variables  $V$ . Using these predicates, users can define temporal properties related to the state of the kernel and the state of the SystemC model. Recall that *Producer* and *Consumer* are two SystemC modules that have two functions *send()* and *receive()*, respectively. We consider the following variables *send\_start*, *send\_done* and *rcv*  $\in V$ . They expose the state of the SystemC model as described in the section above. Assume that we want to express the property “over a period of  $T_1$  time units, *send()* remains blocked until *receive()* has returned within  $T_2$  time units”. This property can be specified with the “until” operator that is given as follows.

$$G_{\leq T_1}(\text{send\_start} \rightarrow (\neg \text{send\_done} U_{\leq T_2} \text{rcv}))$$

## 4 Implementation

We have implemented a SMC-based verification framework [26] which is used to analyze the case study in Section 6. Our implementation contains two main components: a *monitor and aspect-advice generator* (MAG) and a *statistical model checker* (SystemC Plugin). The flow of our framework is depicted in Fig. 1.

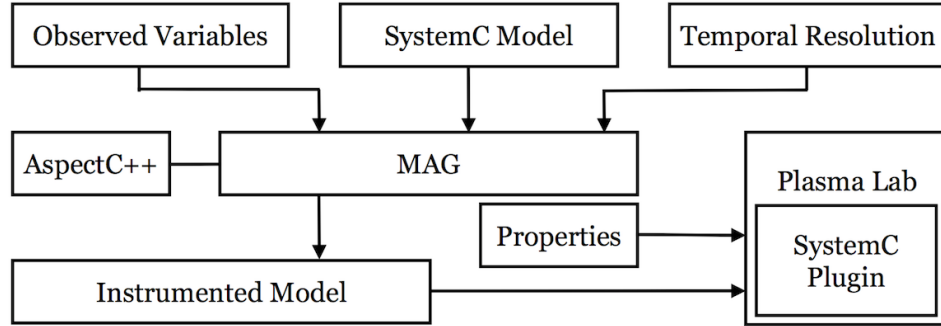


Figure 1: The framework’s flow

### 4.1 Monitor and Aspect-Advice Generator

In principle, the full state can be observed during the simulation of the model. In practice, however, users define a set of variables of interest, called *observed variables*, and only these variables appear in the states of an execution trace. Given a SystemC model, an observed variable is a variable of primitive type (e.g, usual scalar or enumerated type in C/C++). The value of this variable represents a part of the full state of the model. We use  $V_{obs} \subseteq V$  to denote the set of observed variables. Then, the observed execution traces of the model are the projections of the execution traces on  $V_{obs}$ , meaning that for every execution trace  $\omega$ , the

corresponding observed execution trace is  $\omega \downarrow_{V_{obs}}$ . In the following, when we mention about execution traces, we mean observed execution traces.

The implementation of MAG allows users to define a set of observed variables that is used with a temporal resolution to generate a monitor based on the techniques in [29] in order to make an instrumented SystemC model. The instrumented model will produce a set of execution traces of the model. The generated monitor evaluates the set of observed variables at every time point in which an event of the temporal resolution occurs during the SystemC model simulation. The generator also generates an aspect-advice file that is used by AspectC++ [6] to automatically instrument the SystemC model.

## 4.2 SystemC Plasma Lab Plugin

Our statistical model checker is implemented as a plugin of Plasma Lab [2] which establishes an interface between Plasma Lab and the instrumented model being executed by the simulator. In the current version, the communication is done via the standard input and output. The plugin requests new states until the satisfaction of the formula to be verified can be decided, which terminates because the temporal operators are bounded. Similarly, depending on the hypothesis testing algorithms (e.g., sequential hypothesis testing, Monte Carlo simulation, or 2-sided Chernoff bound), the plugin will request new traces from the instrumented model.

## 4.3 Running Verification

Running the verification framework consists of two steps as follows. First, users define a set of observed variables and a temporal resolution in a configuration file, as well as other necessary information. From that information, the generator generates the monitors and aspect-advice that are used by AspectC++ to produce the instrumented SystemC model. In addition, the generator can automatically generate a Plasma Lab project file according to the desired properties. The instrumented model and the generated monitors are compiled together and linked with the SystemC simulation kernel into an executable model in order to make a set of execution traces of the system. In the second step, the plugin of Plasma Lab is used to verify the desired properties. The satisfaction checking of the properties is brought out based on the set of execution traces by executing the instrumented SystemC model and can be done by several hypothesis testing algorithms provided by Plasma Lab. The full implementation of our verification framework including the monitor and aspect-advice generator and the checker can be downloaded on the website of Plasma Lab<sup>2</sup>.

# 5 Modeling Dependability in SystemC

SHLPNs are high-level Petri nets (HLPNs) [24, 14, 21], in which each transition execution has a duration described by an exponential distribution. They are commonly used for modeling distributed systems in order to study their performance and dependability [8, 21, 20]. In this section, we propose an approach for realizing SHLPNs in SystemC such that the semantics is preserved.

## 5.1 Stochastic High-Level Petri Nets

High-level Petri nets provide a compact representation of complex systems. There are many different types of HLPNs that have been proposed in literature such as predicate transition

<sup>2</sup>MAG manual: [https://project.inria.fr/plasma-lab/documentation/tutorial/mag\\_manual/](https://project.inria.fr/plasma-lab/documentation/tutorial/mag_manual/)

nets [7], coloured Petri nets [12] and relation nets [27]. However, in [13, 27] it is proved that one can translate the HLPN of a system in one type into any other type. Due to the intuition and the modeling elegance we consider the predicate transition nets in which the tokens are *coloured* or *typed tokens* [14]. Each place is annotated with a data type. Each place is associated with a *place capacity*  $K$  which bounds for the number of tokens that the place can contain. The arcs are labeled by tuples of token variables. The zero-tuple indicates an ordinary place, meaning a no-argument token. The transitions are annotated with *guard formulas* defined over variables labelling the adjacent arcs and a set of *variable assignments*. A transition is *enabled*, that is may execute or *fire*, whenever 1) all input places carry enough tokens, 2) there is an assignment of tokens to token variables that satisfies the guard formula of the transition, and 3) each output place capacity is sufficient to store the tokens produced by the transition. By firing, a transition removes and adds tokens from/to places according to the expressions labeling the arcs. SHLPNs are high-level Petri nets in which each execution of a transition lasts for a given amount of time, called *firing time*. Firing times are specified by an exponential distribution associated to each transition. Due to the memoryless property of the exponential distribution, this class of SHLPNs is isomorphic to continuous-time Markov chains (CTMCs) as shown in [21]. We consider the example in Fig. 2. This net is a part of the SHLPN of our case study models the

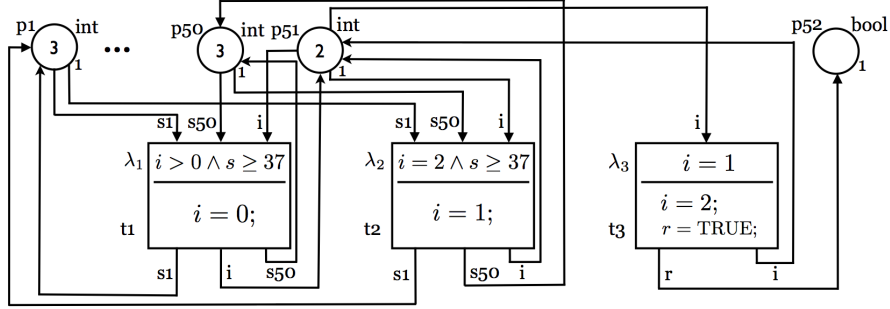


Figure 2: Stochastic high-level Petri net example

reliability of the input processor. The example consists of 51 I/O places with data type `int` ( $p_1$  to  $p_{50}$  represent the number of functional sensors in each group and  $p_{51}$  representing the current status of the processor) and output place  $p_{52}$  with data type `bool` carries a token with the value `true` whenever the processor reboots successfully. Their capacities are 1. Each of transitions  $t_1, t_2$  and  $t_3$  is annotated with a firing time following the exponential distribution with the rate  $\lambda_i, i = 1, \dots, 3$ . These rates may be marking-dependent. With the marking depicted in Fig. 2,  $s_1$  to  $s_{50}$ , and  $i$  are assigned values 3 and 2 respectively, both transitions  $t_1$  and  $t_2$  are enabled to fire that means they conflict, where  $s = \sum_{k=1}^{50} \sigma_k, \sigma_k = 1$  if  $s_k \geq 2$ , otherwise  $\sigma_k = 0$ . Assume that  $t_1$  fires,  $p_1$  to  $p_{50}$  carry the same previous token values,  $p_{51}$  carries the value 0 and  $p_{52}$  is empty.

## 5.2 Connection between HLPNs and Rule-Based Systems

A rule-based system (or production based system) [3] consists of a *working memory* containing known facts, a *production memory* containing rules, and an *inference engine* which matches the rules with the working memory to infer new facts by applying the selected rule among all the applicable rules and the existing facts. The rules syntactically are of the form “*if conditions then actions*”. The conditions are patterns that are checked for the rule activation, called the rule *antecedent*. If the conditions match with the facts, the actions, called the rule *consequent*, are performed. Fig. 3 depicts the main structure of a forward chaining inference algorithm. The

```

1 void Infer(working_memory,
2           production_memory) {
3     rules = Select(working_memory,
4                   production_memory);
5     while(rules ≠ ∅) {
6       rule = SolveConflicts(rules);
7       ApplyRule(rule);
8       rules = Select(working_memory,
9                     production_memory);
10    } //END while
11  }

1 rule_set Select(working_memory,
2                production_memory) {
3     rules = ∅;
4     for rule ∈ production_memory {
5       if Match(rule, working_memory)
6         rules = rules ∪ rule;
7     } //END for
8     return rules;
9  }

```

Figure 3: A forward chaining inference algorithm

main function, **Infer**, executes the rule system until no more rule can be executed. The function **Select** identifies a set of rules that matches the facts in the working memory, according to the **Match** function, which maps the variables appearing into the conditions to some constants from the facts of the working memory. The function *SolveConflicts* selects one of the selected rule to be fired. Finally, the function *ApplyRule* executes the rule actions and updates the working memory.

For every HLPN one can construct a rule-based system, as shown in [31, 4]. The transformation consists of the production memory and working memory constructions. For each transition of the net, a rule is added in which the conditions and actions are defined by the guard formula, the set of variable assignments, the input and output places of the transition. For each input place, the facts that bound the tokens to the variables labeling the arc connecting the place to the transitions are added and the condition expressing that the place carries enough copies of proper tokens, as well as the action of removing tokens from the place are defined for the corresponding rule. For each output place, the action of adding tokens to the place is associated, by specifying the place name and the variables on the arc connecting the transition to the place and the condition expressing that the place's capacity is not exceeded by adding the produced tokens is defined. The actions of adding and removing tokens update the facts in the working memory. The initial working memory is constructed based on the initial marking of the net. In other words, a fact represents a multiset of constants. This set is compatible with the type and capacity of its associated place. The following proposition from [31, 4] shows that the HLPN and its corresponding transformed rule-based system are semantically equivalent.

**Proposition 5.1** *For every high-level Petri net, a rule-based system exists that is semantic equivalent.*

The proof of Proposition 5.1 and the similarities between HLPNs and rule-based systems have been studied in [31, 4, 25] by showing that the rules applied by the inference algorithm represent the reachability set from the initial marking of the initial Petri net. It follows that the inference algorithms can be used to implement the operational semantics of the stochastic high-level Petri nets. In addition, it is shown in [31, 4] that the implementation of HLPNs semantics based on rule-based systems with an improved version of the inference algorithm was described by Forgy [5], is most efficient when the net have large number of places and large number of tokens distributed among the places, as well as when the large number bounded variables labeling the arcs. In the next section, we propose an approach to realize SHLPNs by implementing the inference algorithms in SystemC.

### 5.3 Realization of SHLPNs

We illustrate our realization of the inference algorithm in Fig. 3 with the example in Fig. 2. It is implemented by a SystemC module as shown in Fig. 4, in which the *Infer* function is a



thread process in SystemC. The initialization initialize the working memory by setting all places, the capacity and the initial marking in the constructor method (lines 37 to 40). And a place is implemented as an instance of a template class (i.e., *place<int>*) that contains the facts. The template class has the methods for getting a token value (*get*), storing a token (*mark*), removing a token (*demark*), getting the capacity (*get\_capacity*), and the current number of tokens (*get\_num*) in a place.

Consider a SHLPN with the current marking  $M_j$ , transitions become enabled as usual, i.e., if all input places have sufficiently tokens and the guard formulas are satisfied. However, there is a time, which has to elapse, before an enabled transition fires. We denote the set of all enabled transitions  $t_1, \dots, t_n$  in  $M_j$  by  $E(M_j)$ . Since all firing times are independent exponentially distributed, the minimum firing time is also exponentially distributed with the rate  $\lambda = \sum_{i=1}^n \lambda_i$  and the probability that a given enabled transition, say  $t_k$ , samples the minimum firing time  $Pr(t_k|M_j) = \lambda_k / \sum_{i:t_i \in E(M_j)} \lambda_i$ .

```

1  SC_MODULE(Shlpn) {
2      SC_HAS_PROCESS(Shlpn);
3  public:
4      Shlpn(sc_module_name name, gsl_rng *rnd);
5      //operation of the net
6      void Infer();
7  private:
8      place<int> p[51];
9      place<bool> p52;
10     int i, s[50];
11     bool r;
12     //firing rates
13     double r1, r2, r3, sum_r, ft;
14     //GSL random generator
15     gsl_rng *rnd;
16     //enabled transitions
17     bool e[3];
18     //return enabled transitions
19     int Select();
20     int SolveConflicts();
21     ApplyRule(int rule);
22 };
23 void Shlpn::Infer() {
24     for (int k = 0; k < 3; k++)
25         e[k] = false;
26     sum_r = ft = 0;
27     int rules = Select();
28     while(rules > 0) {
29         int rule = SolveConflicts();
30         ApplyRule(rule);
31         for (int k = 0; k < 3; k++)
32             e[k] = false;
33         sum_r = ft = 0;
34         rules = Select();
35     } //END while
36 }

37 Shlpn::Shlpn(sc_module_name name, gsl_rng
38     *rnd) {
39     //initialization
40     SC_THREAD(Infer);
41 }
42 int Shlpn::SolveConflicts() {
43     int rule;
44     double pr1, pr2, pr3;
45     //probability t1 fires
46     pr1 = e[0] ? r1/sum_r : 0;
47     //probabilities of t2, t3
48     pr2 = e[1] ? r2/sum_r : 0;
49     pr3 = e[2] ? r3/sum_r : 0;
50     //fired transition
51     rule = gsl_discrete({pr1,pr2,pr3},rnd);
52 }
53 void Shlpn::ApplyRule(int rule) {
54     switch (rule) {
55     case 0: //fire t1
56         //elapse firing time
57         ft = gsl_exp(sum_r,rnd);
58         wait(ft,time_unit);
59         for (int k = 0; k < 51; k++)
60             p[k].demark();
61         i = 0;
62         for (int k = 0; k < 50; k++)
63             p[k].mark(3);
64         p[50].mark(i);
65         break;
66     case 1: //fire t2
67     case 2: //fire t3
68     default:
69         break;
70     } // END switch
71 }

```

Figure 4: SystemC code for example in Figure 2

Therefore, the implementations of the functions *SolveConflicts* and *ApplyRule* is done as follows. Given a set of all enabled rules from the function *Select* in Appendix A, *SolveConflicts* (lines 41 to 51) determines a rule to be applied from the set of selected rules using a discrete distribution over  $Pr(t_k|M_j)$  (lines 45 to 50). The firing time before the firing of selected rule in *ApplyRule* (lines 52 to 70), is sampled from the exponential distribution with the rate  $\lambda$  (lines 56 to 57). We employ the implementation of the discrete and exponential distributions from GNU Scientific Library (GSL). The firing time elapsing is simulated by a *wait()* statement with an amount of time equals to the firing time (i.e., measured by the time unit in the simulator).

The implementation of the function *Select* and the place template class are given in Listing



2 and Listing 3.

```

1  int Shlpn::Select() {
2      int s, rules = 0;
3      bool check_p = true;
4      //check t1 is enabled
5      //check all input places have sufficiently tokens
6      for(int k = 0; k < 50; k++) {
7          if (p[k]->get_num() > 0)
8              s[k] = p[k]->get();
9          else {
10             check_p = false;
11             break;
12         }
13     } //END for
14     if (p[50]->get_num() > 0)
15         i = p[50]->get();
16     else check_p = false;
17     //check the guard formula
18     if (check_p) {
19         for (int k = 0; k < 50; k++)
20             if (s[k] >= 2)
21                 s = s + 1;
22         if (s >= 37 && i > 0) {
23             //set t1 is enabled transition
24             e[0] = true;
25             sum_r = sum_r + r1;
26             rules = rules + 1;
27         } //END if
28     } //END if
29     //check t2 is enabled
30     //check t3 is enabled
31     return rules;
32 }

```

Listing 2: SystemC code for the *Select* function

```

1  template <class type> class place {
2  public:
3      type get(int); //get token at index
4      bool mark(type); //store token
5      void demark(int); //delete token at index
6      int get_capacity(); //place capacity
7      int get_num(); //number of tokens
8  };
9
10 template <class type> type place<type>::get(int index) {
11     //code
12 }
13
14 template <class type> bool place<type>::mark(type t) {
15     //code
16 }

```

Listing 3: SystemC code for a place

## 6 Case Study and Results

In this section, our SystemC-realization is used to model the dependability of a large embedded control system. We also demonstrate the use of our verification framework to analyze the resulting model. The number of components in our system makes numerical approaches such as PMC unfeasible.

## 6.1 An Embedded Control System

The case study is closely based on the one presented in [23, 17] but contains much more components. The system, depicted in Fig. 5, consists of an input processor ( $I$ ) connected to 50 groups of 3 sensors (from  $S_1$  to  $S_{50}$ ), an output processor ( $O$ ), connected to 30 groups of 2 actuators (from  $A_1$  to  $A_{30}$ ), and a main processor ( $M$ ), that communicates with  $I$  and  $O$  through a bus. At every cycle, the main processor polls results from the input processor that reads and processes data from the sensors. Based on these results, it elaborates commands to be passed to the output processor which controls the actuators. For instance, the input sensors can measure the fluid level, temperature, or pressure, while the commands sent to actuators could be used for controlling valves.

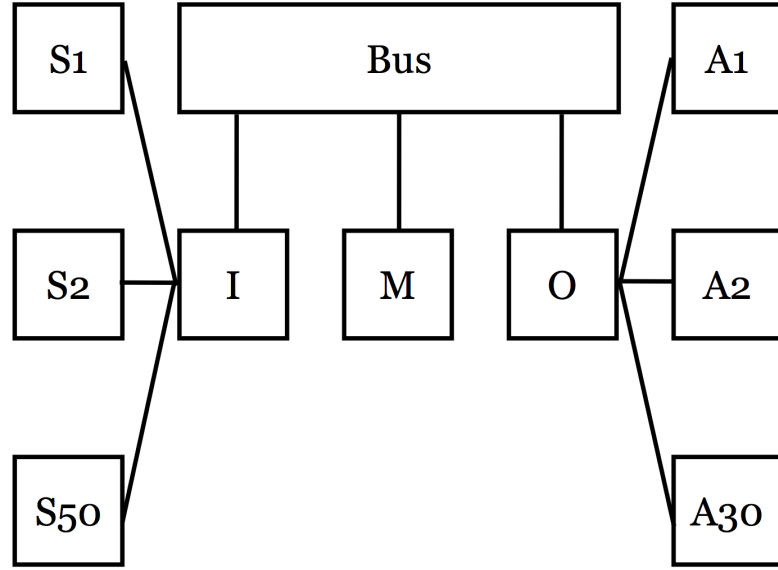


Figure 5: A control system

The reliability of the system is affected by the failures of the sensors, actuators, and processors. The probability of bus failure is negligible, hence we do not consider it. The sensors and actuators are used in 37 – of – 50 and 27 – of – 30 modular redundancies, respectively. Hence, if at least 37 sensor groups are functional (a sensor group is functional if at least 2 of the 3 sensors are functional), the system obtains enough information to function properly. Otherwise, the main processor is reported to shut the system down. In the same way, the system requires at least 27 functional actuator groups to function properly (a actuator group is functional if at least 1 of the 2 actuators is functional). Transient and permanent faults can occur in processors  $I$  or  $O$  and prevent the main processor( $M$ ) to read data from  $I$  or send commands to  $O$ . In that case,  $M$  skips the current cycle. If the number of continuously skipped cycles exceeds the limit  $K$ , the processor  $M$  shuts the system down. When a transient fault occurs in a processor, rebooting the processor repairs the fault. Lastly, if the main processor fails, the system is automatically shut down. The mean times to failure for the sensors, actuators, I/O processors, main processor, and the mean times for the delays are given in Table 1, in which 1 time unit is 30 seconds. A cycle lasts 2 time units, that is 1 minute.

As described above, the system is modeled as a SHLPN. In the net, the places for each sensor group and each actuator group have 4 and 3 different markings, respectively. The places for I/O processors have 3 different marking, and the place for main processor have 2 different marking.

Therefore, the underlying CTMCs for the net has  $\sim 2^{155}$  states comparing to the model in [17] with  $\sim 2^{10}$  states.

## 6.2 Analysis Results

The set of observed variables, the temporal resolution and their meaning are given in Table 1. We define four types of failures: **failure<sub>1</sub>** is the failure of the sensors, **failure<sub>2</sub>** is the failure of the actuators, **failure<sub>3</sub>** is the failure of the I/O processors and **failure<sub>4</sub>** is the failure of the main processor. For example, **failure<sub>1</sub>** is defined as  $(\text{number\_sensors} < 37) \wedge (\text{proci\_status} = 2)$ . It specifies that the number of working sensor groups has decreased below 37 and the input processor is functional, so that it can report the failure to the main processor. We define **failure<sub>2</sub>**, **failure<sub>3</sub>**, and **failure<sub>4</sub>** in a similar way. In our analysis which is based on the one in [17] with  $K = 4$ , we used the Monte Carlo algorithm with 3000 simulations.

Variable name	Meaning	Component	Mean time
<code>number_sensors</code>	Working sensor groups	Sensor	1 month
<code>number_actuators</code>	Working actuator groups	Actuator	2 months
<code>proci_status</code>	Input processor's state	Transient	1 day
<code>proco_status</code>	Output processor's state	Processor	1 year
<code>procm_status</code>	Main processor's state	Timercycle	1 minute
<code>timeout_counts</code>	Number of skipped cycles	Reboot	30 seconds
<code>reward_up</code>	Time in "up" state		
<code>reward_danger</code>	Time in "danger" state		
<code>reward_shutdown</code>	Time in "shutdown" state		
Temporal resolution	Meaning		
<code>tick_notified</code>	Observed variables are evaluated every one time unit		

Table 1: Observed variables and temporal resolution

First, we study the probability that each of the four types of failure eventually occurs during the first  $T$  units of time using the formula  $F_{\leq T}(\text{failure}_i)$ . Fig. 6 plots these probabilities for  $T$  varying from 5 to 30 days of operation. We observe that the probabilities that the sensors and I/O processors eventually fail are higher than the others.

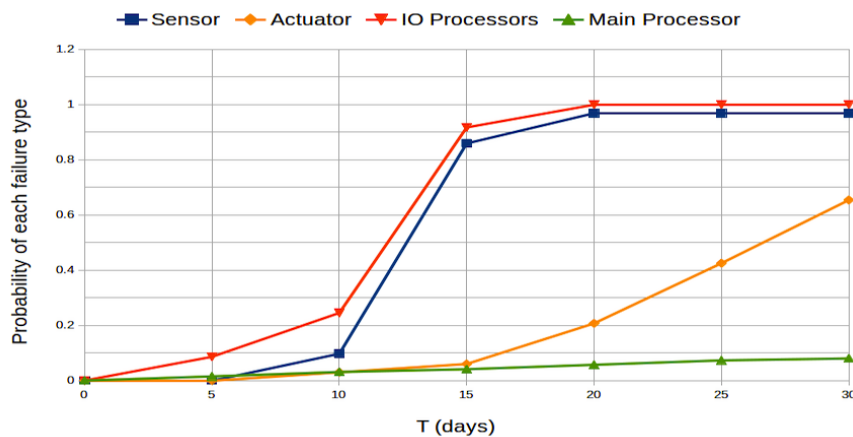


Figure 6: The probability that each of the 4 failure types in the first  $T$  time of operation

For the second part of our analysis, we try to determine which kind of component is more likely to cause the failure of the system. In that frame, it is necessary to determine the probability that

a failure related to a given component occurs before any other failures. The atomic proposition **shutdown**, defined by  $\bigvee_{i=1}^4 \text{failure}_i$  indicates that the system has shut down because one of the failures has occurred. The formula  $\neg \text{shutdown } U_{\leq T} \text{failure}_i$  is true if the failure  $i$  occurs within  $T$  time units and no other failures have occurred before the failure  $i$  occurs, that is if failure  $i$  is the cause of the shutdown. Fig. 7 shows the probability that each kind of failure occurs first over a period of 30 days of operation. It is obvious that the sensors are likelier to cause a system shutdown. At  $T = 20$  days, it seems that we reached a stationary distribution indicating for each kind of component the probability that it is responsible for the failure of the system.

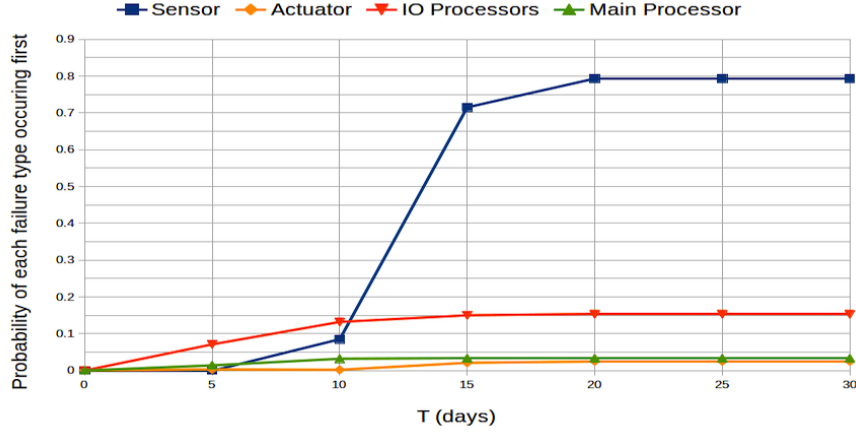


Figure 7: The probability that each of the 4 failure types is the cause of system shutdown in the first  $T$  time of operation

For the third part of our analysis, we divide the states of system into three classes: “up”, where every component is functional, “danger”, where a failure has occurred but the system has not yet shut down (e.g., the I/O processors have just had a transient failure but they have rebooted in time), and “shutdown”, where the system has shut down [17]. We aim to compute the expected time spent in each class of states by the system over a period of  $T$  time units. To this end, we add in the model, for each class of state  $c$ , a variable **reward\_c** that measures the time spent in the class  $c$ . The formula  $X_{\leq T} \text{reward\_c}$  returns the mean value of **reward\_c** after  $T$  time units of execution over the 3000 traces considered. The results are plotted in Fig. 8.

Finally, we approximate the number of input, output processor reboots which occur and the number sensor groups, actuator groups that are functional over time by computing the expected values of random variables that count the number of reboots, functional sensor and actuator groups. The results are plotted in Fig. 9 and Fig. 10. It is obvious that the number of reboots of both processors doubles the number of reboots of each processor since they have the same models.

## 7 Related Work and Conclusion

Some work has been carried out for dependability analysis with PMC, for example, the dependability analysis of control system with PRISM [17]. PRISM supports construction and analysis of Markov chains. For example, the exact probabilities in our case study can be computed by PRISM for the small system with one sensor group and one actuator group. However, the main drawback of this approach is that when it deals with real-world large size systems which make the PMC technique is unfeasible. That means the state explosion likely occurs, even with some

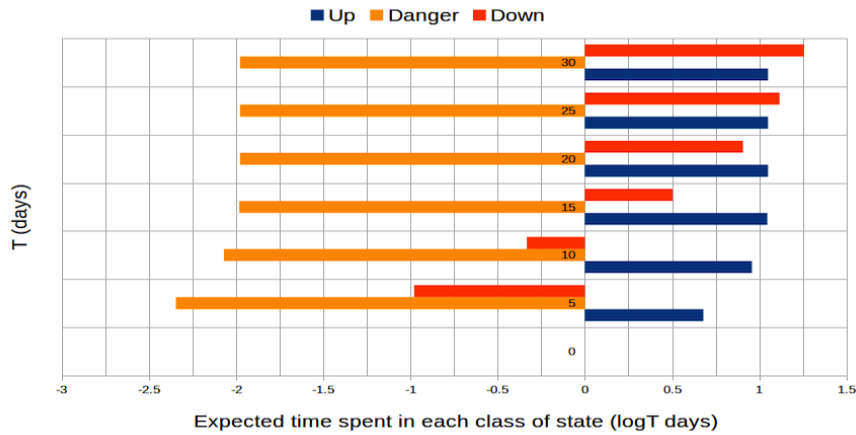


Figure 8: The expected amount of time spent in each of the states: “up”, “danger” and “shutdown”

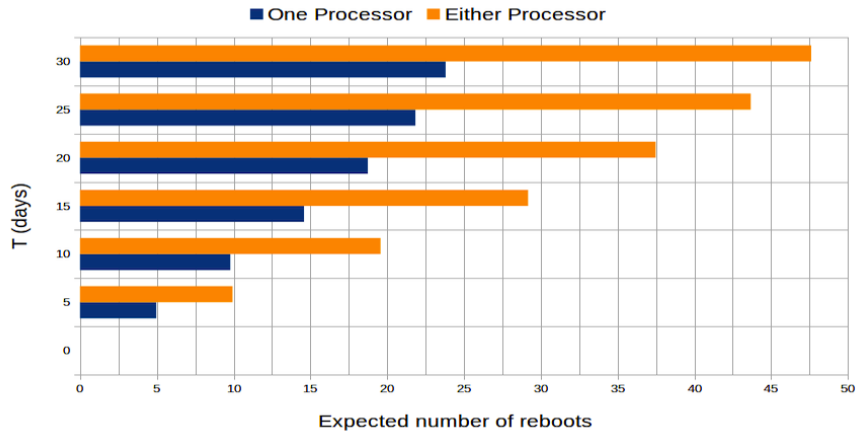


Figure 9: Expected number of reboots that occur in the first  $T$  time of operation

abstraction, i.e., symbolic model checking with *Ordered Binary Decision Diagrams* (OBDDs), is applied.

Tabakov et al. [29] proposed a framework for monitoring temporal SystemC properties. This framework allows users express the verifying properties by fully exposing the semantics of the simulator as well as the user-code. They extend LTL by providing some extra primitives for stating the atomic propositions and let users define a much finer temporal resolution. Their implementation consists of a modified simulation kernel, and a tool to automatically generate the *monitors* and aspect advices for applying *Aspect Oriented Programming* (AOP) [6] to instrument SystemC programs automatically.

This paper presents the first attempt to analyze the dependability of computer-based control systems using statistical model checking, in which the dependability of the systems is modeled by a SystemC-realization of stochastic high-level Petri nets. In comparison to the probabilistic model checking, our approach allows users to handle large industrial systems as well as to expose a rich set of user-code primitives by automatically instrumenting the SystemC code with AspectC<sup>++</sup>.

Currently, we consider an external library as a “black box”, meaning that we do not consider the states of external libraries. Thus, arguments passed to a function in an external library

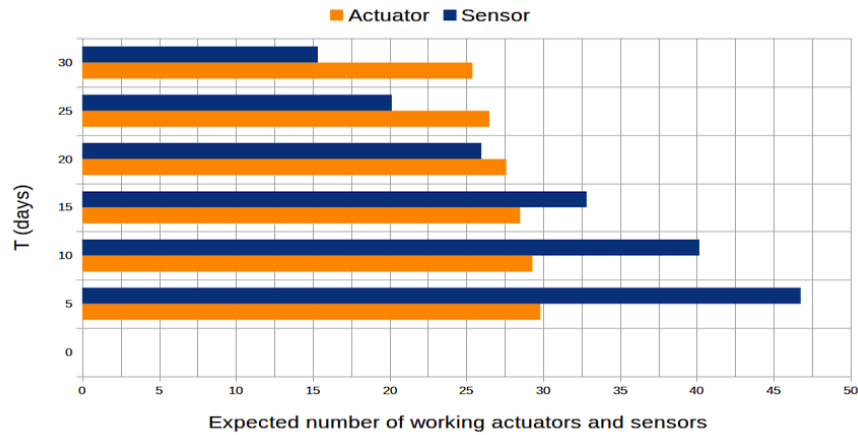


Figure 10: Expected number of functional sensor and actuator groups in the first  $T$  time of operation

cannot be monitored. For future work, we would like to allow users to monitor the states of the external libraries with the future version of AspectC<sup>++</sup>. We also plan to apply statistical model checking to verify temporal properties of SystemC-AMS (Analog/Mixed-Signal) and make the improved version of the current naive inference algorithm implementation based on the RETE algorithm in [5].

## References

- [1] Accellera. <http://www.accellera.org/downloads/standards/systemc>.
- [2] B. Boyer, K. Corre, A. Legay, and S. Sedwards. Plasma lab: A flexible, distributable statistical model checking library. In *QEST'13*, pages 160–164, 2013.
- [3] L. Brownston and al. Programming expert systems in ops5: An introduction to rule-based programming. In *Adisson-Wesley*, 1985.
- [4] D. D. Burdescu and M. Brezovan. Algorithms for high level petri nets simulation and rule-based systems. In *Available at http://software.ucv.ro/mbrezovan/Sibiu.pdf*, 2013.
- [5] C. Forgy. Rete: A fast algorithm for many pattern/many objects. In *Artificial Intelligence*, 1982.
- [6] A. Gal, W. Schroder-Preikschat, and O. Spinczyk. Aspectc<sup>++</sup>: Language proposal and prototype implementation. In *OOPSLA'01*, 2001.
- [7] J. Genrich and K. Lautenbach. The analysis of distributed systems by means of predicate transition nets. In *LNCS*, volume 70, pages 123–146, 1979.
- [8] A. Goyal and et al. Probabilistic modeling of computer system availability. In *Annals of Operations Research*, volume 8, pages 285–306, 1987.
- [9] T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, USA, 2002.

- [10] H. Hermanns, B. Watcher, and L. Zhang. Probabilistic cegar. In *CAV'08*, volume 5123, pages 162–175. LNCS, Springer, 2008.
- [11] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS'06*, volume 3920, pages 441–444. LNCS, Springer, 2006.
- [12] K. Jensen. Coloured petri nets and the invariant method. In *Theoretical Comp. Sci.*, volume 14, pages 317–336, 1981.
- [13] K. Jensen. High-level petri nets. In *Informatik-Fachberichte*, volume 66, pages 166–180, 1983.
- [14] K. Jensen and L. Kristensen. Coloured petri nets: Modeling and validation of concurrent systems. In *Springer Verlag*, 2009.
- [15] S. Jha, E. Clarke, C. Langmead, A. Legay, A. Platzer, and P. Zuliani. A bayesian approach to model checking biological systems. In *CMSB'09*, volume 5688, pages 218–234. LNCS, Springer, 2009.
- [16] J. Katoen, E. Hahn, H. Hermanns, D. Jansen, and I. Zapreev. The ins and outs of the probabilistic model checker mrmc. In *QEST'09*. IEEE CS Press, 2009.
- [17] M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. In *Control Engineering Practice*, volume 15(11), pages 1427–1434. Elsevier, 2007.
- [18] A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *RV'10*, volume 6418, pages 122–135. LNCS, Springer, 2010.
- [19] R. Lipsett, C. Schaefer, and C. Ussery. *VHDL: Hardware description and design*. Kluwer Academic Publishers, 1993.
- [20] M. Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. In *ACM Trans. Comp. Systems*, volume 2(2), pages 93–122, 1984.
- [21] M. Molloy. Performance analysis using stochastic petri nets. In *IEEE Trans. Comp*, volume C-31(9), pages 913–917, 1982.
- [22] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of systemc. In *DATE 2001*, pages 64–70, 2001.
- [23] J. Muppala, G. Ciardo, and K. Trivedi. Stochastic reward nets for reliability prediction. In *Communications in Reliability, Maintainability and Serviceability*, volume 1(2), pages 9–20, 1994.
- [24] T. Murata. Petri nets: Properties, analysis and applications. In *In Proceedings of the IEEE*, volume 77(4), pages 541–580, 1989.
- [25] T. Murata and D. Zhang. A predicate-transition net model for parallel interpretation of logic programs. In *IEEE Transactions on Software Engineering*, volume 14(1), 1988.
- [26] V. C. Ngo, A. Legay, and J. Quilbeuf. Dynamic verification of systemc with statistical model checking. In *Research Report - RR-8644*. INRIA Rennes - Bretagne Atlantique, ESTASYS, 2014.

- 
- [27] W. Reisig. Petri nets with individual tokens. In *Informatik-Fachberichte*, volume 66, pages 229–249, 1983.
  - [28] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *CAV'05*, volume 3576, pages 266–280. LNCS, Springer, 2004.
  - [29] D. Tabakov and M. Vardi. Monitoring temporal systemc properties. In *Formal Methods and Models for Codesign*, pages 123–132. IEEE, 2010.
  - [30] D. Thomas and P. Moorby. The verilog hardware description language. In *Springer. ISBN 0-3878-4930-0*, 2008.
  - [31] R. Valette and B. Bako. Software implementation of petri nets and compilation of rule-based systems. In *Advances in Petri Nets*, 1990.
  - [32] H. Younes. Ymer: A statistical model checker. In *CAV'05*, volume 3576, pages 429–433, 2005.





**RESEARCH CENTRE  
RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu  
35042 Rennes Cedex

Publisher  
Inria  
Domaine de Volveau - Rocquencourt  
BP 105 - 78153 Le Chesnay Cedex  
[inria.fr](http://inria.fr)

ISSN 0249-6399